Variable Structure System Simulation via Predefined Acausal Components

Andrea Neumayr[1] Martin Otter[1]

[1]German Aerospace Center (DLR), Institute of System Dynamics and Control (SR), Germany,
{andrea.neumayr,martin.otter}@dlr.de

Abstract

This article outlines a new approach of the experimental open-source modeling and simulation system Modia to simulate systems where the number of variables and equations can be changed after compilation and also during simulation, without having to re-generate and re-compile the code. Details are given for heat transfer in an insulated rod, where the discretisation of the rod is completely hidden from the symbolic engine. It is discussed how this approach could also be used in a future version of Modelica and/or FMI. Furthermore, this feature is also used in various variants to speed up collision handling in 3D mechanical systems. For example, by rigidly fixing an object after it has been gripped, with or without calculating the elastic response, and thereby dynamically changing the number of degrees of freedom.

Keywords: Modia, Julia, multibody, segmented simulation, heat transfer, collision handling

1 Introduction

Modia (Elmqvist et al. 2021) is an experimental, open source modeling and simulation system to develop new approaches to overcome the limitations of declarative, equation-based modeling languages such as Modelica (Modelica Association 2023). Modia is implemented with the powerful Julia programming language (Bezanson et al. 2017). It consists of a set of Julia packages, in particular of Modia.jl[1] for equation-based modeling à la Modelica and of Modia3D.jl[2] for modeling of multibody systems.

Neumayr and Otter (2023) extend Modia to process so called *predefined acausal components*[3]. These model components consist of a (usually small) set of Modia equations in which Julia functions are called that contain the core variables and equations of the components. These variables and equations can appear and disappear during simulation, without re-generation and re-compilation of code and without knowing in advance which model equations are utilized during such a simulation.

In contrast to this new approach, all previous proposals for systems with variable structure must either know in advance the entire models for all modes and switch between these models during simulation, (e.g., Mehlhase 2014; Mattsson, Otter, and Elmqvist 2015; Tinnerholm, Pop, and Sjölund 2022). Or the entire model is newly processed and code is re-generated and re-compiled (or interpreted) whenever the equation structure is changed[4], (e.g., Zimmer 2010; Tinnerholm, Pop, and Sjölund 2022).

In this article, the novel approach in Modia for modeling predefined acausal components is demonstrated with 1D heat transfer in a rod, where the number of discretization nodes can be changed before simulation start without re-compilation. It would also be possible to change the number of discretization nodes during simulation.

Modia3D is a more complex predefined acausal component. It was recently extended to cope with variable structure systems where the number of degrees of freedom can change during simulation, without re-compilation. The core part of this article discusses how this new feature is used to improve collision handling as an extension to the elastic response calculation introduced in (Neumayr and Otter 2019).

Elmqvist et al. (2021, Section 2) describe the Modia Language, and Neumayr and Otter (2023, Appendix A) provide a short overview of it. A more detailed explanation is available in the Modia Tutorial[5].

2 Predefined Acausal Components

Neumayr and Otter (2023) introduce predefined acausal components which are based on a proposal of Elmqvist (2022): The equations of an acausal component are split into causal and acausal partitions. The intuition is that the causal partition is always evalu-

[1]https://github.com/ModiaSim/Modia.jl, v0.12.0, visited on 2023-06-13

[2]https://github.com/ModiaSim/Modia3D.jl, v0.12.0, visited on 2023-06-13

[3]Neumayr and Otter (2023) refers to these components as acausal built-in components. We decided to rename them to predefined acausal components to be more descriptive.

[4]Generated compiled code maybe cached.

[5]https://modiasim.github.io/Modia.jl/stable/, visited on 2023-06-13

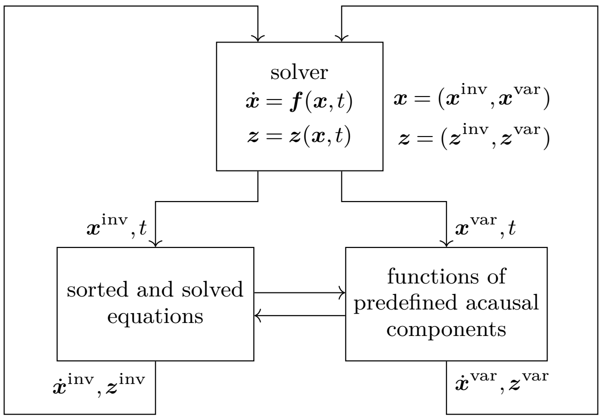
ated in the same order, regardless of how the component is connected with other components. This partition is sorted, explicitly solved for the unknowns, and implemented with one or more functions. In contrast, sorting and solving of the acausal partition depends on the actual connection of the component. This partition is kept as a set of equations. Note that, the figures and some text fragments used below in this section are from Neumayr and Otter (2023)

In Neumayr and Otter (2023), various variants of this basic approach are discussed. In particular, (a) the variables computed in the causal partitions can either be still visible in the equation part as proposed by Elmqvist (2022) or (b) a large part of these variables is hidden in the functions and do no longer appear in the equation part. Variant (a) has the advantage, that index reduction is still possible by differentiating the functions of the causal partitions. Variant (b) has the advantage that the causal partition, in particular the number of its variables and equations, can be changed after compilation and during simulation. Variant (b) has the drawback that index reduction is no longer possible for the causal partitions. Index reduction in the acausal partitions is still possible and is sufficient in many practical cases. However, it is not possible to use a predefined, acausal component as an inverse model if implemented with variant (b). In Modia and Modia3D variant (b) is used.

In Figure 1 the communication structure between the solver, the sorted and solved equations and the functions[6] of the causal partitions are shown: The variables of the solver (state vector $\boldsymbol{x}$ and the vector of event indicators $\boldsymbol{z}$) are split into an invariant and a variant part: $\boldsymbol{x} = (\boldsymbol{x}^{\text{inv}}, \boldsymbol{x}^{\text{var}})$ and $\boldsymbol{z} = (\boldsymbol{z}^{\text{inv}}, \boldsymbol{z}^{\text{var}})$. The dimensions of the invariant parts are fixed before the simulation starts. The dimensions of the variant parts, which are contained in the functions of the causal partitions, can change at events during simulation. Since $\boldsymbol{x}^{\text{var}}, \boldsymbol{z}^{\text{var}}$ are communicated directly between the functions and the solver, the symbolic processing of the equation part of a model is not affected by these variables. Therefore, these variables can in principle be changed at event times - variables can be added or can disappear.

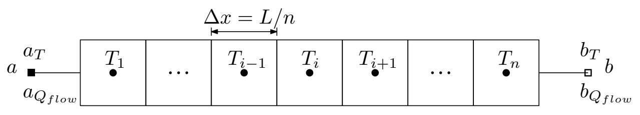
This basic approach is demonstrated using the predefined acausal component of Figure 2, which models heat transfer in a rod with an insulated surface. On the left and right sides of the rod, thermal connectors a, b are present (called `port_a, port_b` in Listing 1) with potential variables a_T, b_T (temperatures) and flow variables $a_{Q_{\text{flow}}}, b_{Q_{\text{flow}}}$ (heat flow rates). The partial differential equation, which mathematically describes the heat transfer in one dimension is discretized in space by volumes $V_i = \Delta x \cdot A$ of equal lengths Δx and identical areas A. In the center of volume i, a temperature T_i is defined, leading to a temperature vector $\boldsymbol{T} = [T_1, T_2, \ldots, T_n]$.

[6]These functions have a memory and are therefore no mathematical functions.



solver
$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, t)$
$\boldsymbol{z} = \boldsymbol{z}(\boldsymbol{x}, t)$
$\boldsymbol{x} = (\boldsymbol{x}^{\text{inv}}, \boldsymbol{x}^{\text{var}})$
$\boldsymbol{z} = (\boldsymbol{z}^{\text{inv}}, \boldsymbol{z}^{\text{var}})$
$\boldsymbol{x}^{\text{inv}}, t$
$\boldsymbol{x}^{\text{var}}, t$
sorted and solved equations
functions of predefined acausal components
$\dot{\boldsymbol{x}}^{\text{inv}}, \boldsymbol{z}^{\text{inv}}$
$\dot{\boldsymbol{x}}^{\text{var}}, \boldsymbol{z}^{\text{var}}$

Figure 1. Communication between the solver, the sorted and solved equations, and the functions of the predefined acausal components. The state vector $\boldsymbol{x}$ and the event indicators $\boldsymbol{z}$ are split into an invariant and a variant part: $\boldsymbol{x} = (\boldsymbol{x}^{\text{inv}}, \boldsymbol{x}^{\text{var}})$, $\boldsymbol{z} = (\boldsymbol{z}^{\text{inv}}, \boldsymbol{z}^{\text{var}})$. The variant parts consist of the states defined and used in the causal partitions of all predefined acausal components. The dimensions of the invariant parts are fixed before simulation starts. The dimensions of the variant parts can change at events during simulation.



$$Q_{\text{flow},i} = \lambda \frac{A}{\Delta x} \begin{cases} 2(a_T - T_1) & i = 0 \\ T_i - T_{i+1} & i = 1, \ldots, n-1 \\ 2(T_i - b_T) & i = n \end{cases}$$

$$\begin{aligned} a_{Q_{\text{flow}}} &= Q_{\text{flow},0} \\ b_{Q_{\text{flow}}} &= -Q_{\text{flow},n} \\ \varrho c A \Delta x \dot{T}_i &= Q_{\text{flow},i-1} - Q_{\text{flow},i} \quad i = 1, \ldots, n \\ T_i(t = t_0) &= T_0 \end{aligned}$$

Figure 2. Space discretized partial differential equation of one-dimensional heat transfer in a rod with an insulated surface. It is defined with parameters L (length of rod), n (number of volumes), A (area), ϱ (density), c (specific heat capacity), λ (thermal conductivity), T_0 (initial value in each volume), states T_i (temperatures in the center of each volume), thermal connectors a, b with potential variables a_T, b_T (temperatures), and flow variables $a_{Q_{\text{flow}}}, b_{Q_{\text{flow}}}$ (heat flow rates).

Listing 1. Simple usage of insulated rod `InsulatedRod2` with one-dimensional heat-transfer. On the left side it is connected with a fixed temperature source `FixedHeatFlow` with $T = 220\,°\text{C} = 493.15\,\text{K}$, and on the right side with a fixed heat flow source `FixedHeatFlow` with $Q_{\text{flow}} = 0$.

```
using Modia
include(
  "$(Modia.path)/models/HeatTransfer.jl")

# Temp. source - rod - heat flow source
HeatedRod = Model(
  # temperature source
  fixedT = FixedTemperature |
    Map(T=493.15),

  # heat flow source (Q_flow=0)
  fixedQflow = FixedHeatFlow,

  # insulated rod with 5 volumes
  rod = InsulatedRod2 |
    Map(L=1.0, T0=273.15, nT=5),

  # connecting the components
  equations = :[
    connect(fixedT.port, rod.port_a),
    connect(rod.port_b, fixedQflow.port)]
)

# generate and compile Julia code
heatedRod = @instantiateModel(HeatedRod)

# change to 8 volumes and simulate model
simulate!(heatedRod, stopTime = 1e5,
  merge=Map(rod = Map(n=8))

# plot temperatures
plot(heatedRod,
  ("fixedT.port.T", "rod.T"))
```

In Listing 1, a Modia model is shown with a predefined acausal component `InsulatedRod2` of the rod[7]. Its left thermal connector `port_a` has a fixed temperature source `FixedTemperature`. Its right thermal connector `port_b` has a fixed heat-flow source `FixedHeatFlow` with the default zero heat-flow rate. This means that, the rod is completely insulated on the right side and has a fixed temperature on the left side. Note that, `A|B` merges model or parameters `B` with model `A`. Command `@instantiateModel(HeatedRod)` symbolically processes this model and generates Julia code that is translated to executable code. The `simulate!` statement changes the discretization, and thus the dimension of the temperature vector $\boldsymbol{T}$, from 5 to 8 volumes before simulation starts without a new translation. The plot of Figure 3 is generated with `plot(heatedRod, ...)`, displaying the temperatures at the temperature source and in the rod volumes.

[7]This model can be found in Modia, v0.12.0, models/HeatTransfer.jl.

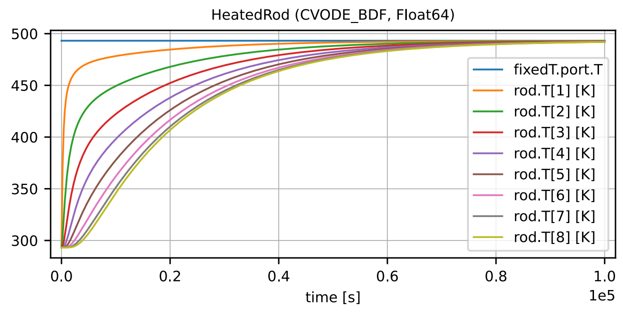


Figure 3. Plot of temperatures of heated rod model.

Listing 2. Modia definition of `InsulatedRod2` model.

```
include("HeatTransfer/InsulatedRod2.jl")

InsulatedRod2 = Model(;
  # Called once before symb. processing
  _buildFunction= Par(functionName =
    :(buildInsulatedRod2!)),

  # Called once before new sim. segment
  _initSegmentFunction=Par(functionName=
    :(initSegmentInsulatedRod2!)),

  # Parameters
  L      = 1.0,
  A      = 0.0004,
  rho    = 7500.0,
  lambda = 74.0,
  c      = 450.0,
  T0     = 293.15,
  nT     = 1,

  # Connectors
  port_a  = HeatPort,
  port_b  = HeatPort
)
```

The implementation of the `InsulatedRod2` model is shown in Listing 2. To start with, file `InsulatedRod2.jl` is included containing the definition of a Julia struct holding the data and the local variables of the component, as well as some Julia functions. More details are given below. A standard Modia definition of the model is then given defining the parameters and connectors of the component. Contrary, to a standard Modia component, no equations are present. For simplicity, no units are used in this model and its associated functions. However, the actual implementation of the component in Modia supports units. The component is a predefined acausal component model, since the following special parameters are provided at the beginning of the model, defining functions to be used for symbolic processing and during simulation:

- `_buildFunction`: Before symbolic processing begins, the hierarchical dictionary of the root model to be compiled is traversed and function `buildInsulatedRod2`, defined from `_buildFunction`, is executed for each sub-model it contains. This function (a) defines additional model variables and equations that are merged with the corresponding model and (b) returns an instance of the Julia structure, which acts as the internal memory of the component.

- `_initSegmentFunction`: This function is called by the simulation engine before the root model is initialized and at each `FullRestart` event before the root model is re-initialized at a new simulation segment. In both cases, all local variables of the predefined acausal component model (including states and zero-crossing functions) must be redefined, as well as initial values for newly defined states.

Listing 3. `_buildFunction` function definition.

```
# Called once before symb. processing
function buildInsulatedRod2!(model,..,ID)
  model = model | Model(
    # Instance of an internal struct
    insRod   = Var(hideResult=true),

    # Dummy return argument
    success = Var(hideResult=true),

    equations = :[
      # copy states into insRod
      insRod = openInsulatedRod!(
          instantiatedModel, $ID)

      # equations at the boundaries
      port_a.Q_flow = getGe2(insRod)*
        (port_a.T - getT1(insRod))
      port_b.Q_flow = getGe2(insRod)*
        (port_b.T - getTend(insRod))

      # compute der(T)
      success =
        computeInsulatedRodDerivatives!(
        instantiatedModel, insRod,
        port_a.T, port_b.T)
    ]
  )
  return (model, InsulatedRodStruct())
end
```

In Listing 3, the implementation of function `buildInsulatedRod2` is shown. In this function, the model instance (of the actual `InsulatedRod2` component) is merged with additional model definitions consisting of two variables and several equations. It returns the merged model. Additionally, the internal memory of the component is instantiated with `InsulatedRodStruct()` and is also returned by `buildInsulatedRod2!`. This internal memory is later identified by the unique identifier `ID`, which is specified in the function call. Function call `openInsulatedRod!` in the equation section copies the rod temperatures $\boldsymbol{T}$ from the state vector of the simulation engine into the `InsulatedRodStruct` memory and returns a reference to it as `insRod`. The argument list of this function call includes the unique identification `ID` of the predefined acausal component. It is provided when `buildInsulatedRod2!` is called. `$ID` is inside an Abstract Syntax Tree, due to `:[...]` and `$` inserts the actual (literal) value at this place. In Julia terminology this is called "interpolation". The `insRod` reference is then used in subsequent function calls, for example, to retrieve the value of the first temperature node with `getT1(insRod)`. This value is used in an equation to calculate the heat flow from `port_a` to the first internal node. Finally, `computeInsulatedRodDerivatives!` computes the derivatives of the temperatures and copies them into the state derivative vector of the simulation engine. As can be seen, the equation section is independent from the number of discretization elements `nT`. Therefore, the number of these discretization elements can be changed without re-generation and re-compilation.

Listing 4. `_initSegmentFunction` definition.

```
# Called once before new sim. segment
function initSegmentInsulatedRod2!(
        m, path, ID, parameters)
  insRod=get_instantiatedSubmodel(m,ID)

  if isFirstInitialOfAllSegments(m)
    initFirstSegmentInsulatedRod2!(
      insRod; parameters...)
  end

  # Define new states and state derivat.
  insRod.T_startIndex =
    new_x_segmented_variable!(m,
      path*".T", path*".der(T)",
      insRod.T, "K")
  return nothing
end
```

The implementation of the `_initSegmentFunction` is shown in Listing 4. This function is called before a new simulation segment is initialized. The first statement inquires the reference `insRod` of the internal memory of the component. Before the first simulation segment, the (evaluated) `parameters` are stored in `insRod`. Furthermore, some dependent parameters are computed and also stored in this memory. Finally, `new_x_segmented_variable` is called to define the name of a new state, its derivative and its unit together with the initial value of `insRod.T` which is the current value of vector `T`. It is initialized with parameter `T_init` in `initFirstSegmentInsulatedRod2!`. Note that, even if the `InsulatedRod2` component always uses the same definition, the states must be newly defined for each new simulation segment.

Listing 5. Function to compute state derivatives

```
# Inquire values from InsulatedRodStruct
getT1(   insRod) = insRod.T[1]
getTend(insRod) = insRod.T[end]
getGe2( insRod) = insRod.Ge2
                  # = 2*lambda*A/dx

# Compute and copy state derivatives
function computeInsulatedRodDerivatives!(
    m, insRod, Ta, Tb)
  T = insRod.T
  k = insRod.k   # = lambda/(c*rho*dx*dx)
  for i in 1:length(T)
    insRod.der_T[i] =
      k*(T_grad1(T,Ta,i) -
      T_grad2(T,Tb,i))
  end
  copy_der_x_segmented_value_to_state(m,
    insRod.T_startIndex, insRod.der_T)
  return true
end
```

In Listing 5, the most important remaining functions are shown. The state derivatives are computed and copied to the state derivative vector of the simulation engine with `computeInsulatedRodDerivatives!`.

Listing 6. Sketch to implement InsulatedRod model as Modelica ExternalObject.

```
class InsulatedRodObject
  extends ExternalObject;

  function constructor
    input Real L, A, rho, lambda, d, T0;
    input Integer nT;
    output InsulatedRodObject insRod;
    external "C" insRod =
      openInsulatedRod(L,A,rho,lambda,
        d,T0,nT);
  end constructor;

  function destructor
    input InsulatedRodObject insRod;
    external "C"
      closeInsulatedRod(insRod);
    end destructor ;
end InsulatedRodObject;

model InsulatedRod
  import H=Modelica.Thermal.HeatTransfer;
  parameter Real    L;
  parameter Real    A;
  parameter Real    rho;
  parameter Real    lambda;
  parameter Real    T0;
  parameter Integer nT=1;

  H.Interfaces.HeatPort_a port_a;
  H.Interfaces.HeatPort_b port_b;
protected
  InsulatedRodObject insRod =
    InsulatedRodObject(
       L,A,rho,lmbda,T0,nT);
  Boolean success;
equation
  // equations at the boundaries
  port_a.Q_flow = getGe2(insRod)*
    (port_a.T-getT1(insRod));
  port_b.Q_flow = getGe2(insRod)*
    (port_b.T-getTend(insRod));

  // compute der(T)
  success =
    computeInsulatedRodDerivatives(
      insRod,port_a.T,port_b.T)
end InsulatedRod;
```

Note that, a similar approach could be implemented in Modelica with reasonable effort: The simplest implementation would be to use External Objects and add additional utility functions (Modelica Association 2023, Section 12.9.6–12.9.7). These are equivalent to the utility functions of Modia, e.g., to add variables at events. These utility functions would directly communicate with the underlying simulation engine. If they are available, the Modia example of the insulated rod could be implemented as outlined in Listing 6. The main benefit would be that the number of temperature nodes can be changed after translation and that the Modelica model consists essentially of three scalar equations. These equations are independent from the number of temperature nodes. The drawback is that functions `openInsulatedRod`, `closeInsulatedRod`, `getGe2`, `getT1`, `getTend`, `computeInsulatedRodDerivatives` need to be implemented in C. Note that these would be simple C-functions. For example, `computeInsulatedRodDerivatives` could be implemented as shown in Listing 7.

Listing 7. C-function to compute state derivatives

```
int computeInsulatedRodDerivatives(
    struct M *m, struct InsRod *insRod,
    double Ta, double Tb) {
  double* T   = insRod->T;
  double* der_T = insRod->der_T;
  double  k   = insRod->k;
  int     nT  = insRod->nT;
  double  k1  = insRod->k1;

  der_T[1] =
    k1*(2*(Ta-T[1])-(T[1]-T[2]));
  der_T[nT] =
    k1*(T[nT-1]-T[nT]-2*(T[nT]-Tb));
  for (i=2; i < nT-1; ++i) {
    der_T[i] =
      k1*(T[i+1]-T[i]-(T[i]-T[i-1]));
  }
  copy_der_x_segmented_value_to_state(m,
    insRod->T_startIndex, der_T);
  return 0;
}
```

The eFMI standard (Functional Mockup Interface for embedded systems, (Lenord et al. 2021)) defines an intermediate language GALEC to transform acausal models to production code. GALEC is basically a very small subset of the Modelica language with some extensions as needed for embedded systems. The extension also includes a simple form of member functions. If such member functions were supported in Modelica, the implementation of predefined acausal components could be done completely in the Modelica language and without External Objects or C-code.

Modelica models can be exported in FMI format (Modelica Association 2022). This includes Modelica models with External Objects. The FMI standard communicates the values of variables explicitly with setter and getter function calls. In principle, it would be possible to add another variable type to the FMI standard, where internal variables (including states) are communicated directly to the solver and no longer via the setter/getter function calls. If such an enhancement were available, the causal partition part of the Modelica model of Listing 6 could be transformed to an FMU, where the node temperatures would no longer be communicated via the FMI setter/getter functions and would no longer be visible in the environment in which the FMU is used. This would have the great advantage that the number of variables and equations can be changed during the simulation.

3 Segmented Simulation and Collision Handling

Modia3D is a multibody tool for 3D mechanical systems implemented as a predefined acausal component of Modia according to section 2. Modia3D is targeted for solvers with adaptive step size control to compute results close to real physics including collision handling using the Minkowski Portal Refinement (MPR) algorithm (Snethen 2008; Neumayr and Otter 2017) and collision response for elastic contacts (Hertz 1896; Flores et al. 2011; Neumayr and Otter 2019). Modia3D has a very flexible and modular design pattern. It is extended (since v0.12.0) to cope with variable structure systems where the number of degrees of freedom (DoF) can change during simulation, without re-compilation.

Modia3D offers two kinds of joints: The first kind of joints contains Modia equation sections with invariant variables, including invariant states. These invariant elements are visible for Modia and cannot be removed or added during simulation. The interface to the Modia3D functionality is designed to define differential equations only on the Modia side in Modia equation sections, so that state constraints can be defined and index reduction can be performed on invariant states. The joints of the second kind define variant variables, including variant states, which are visible only in the Modia3D predefined acausal component. These joints can be added or removed during simulation. For example, an Object3D has an optional keyword `fixedToParent` with a default value of true. In this case, the Object3D is rigidly connected to its parent Object3D. This means it has zero degrees of freedom. If the value is set to false, the Object3D is allowed to move freely with respect to its parent, meaning it has 6 degrees of freedom and 12 variant states. At events, keyword `fixedToParent` can be changed from false to true and vice versa. Neumayr and Otter (2023, Table 2) define Modia3D actions which modify the second (variant) kind of joints and trigger structural changes during simulation, e.g., `actionAttach`, `actionReleaseAndAttach`, `actionRelease`, `actionDelete`. The new states (joints) added during simulation with e.g., `actionRelease` are initialized based on the last known position, velocity, acceleration and rotation. All remaining states are re-initialized with their last known values. Based on that, the internal 3D structure is rebuilt and executed until another action for a structural change is triggered. This restructuring is performed with dynamic data structures and is extremely fast (< 1 ms).

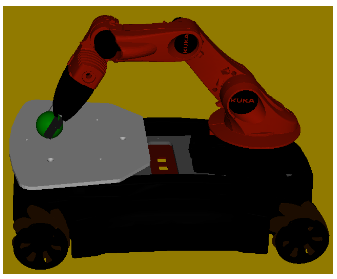


Figure 4. YouBot gripping or releasing a sphere on a plate.

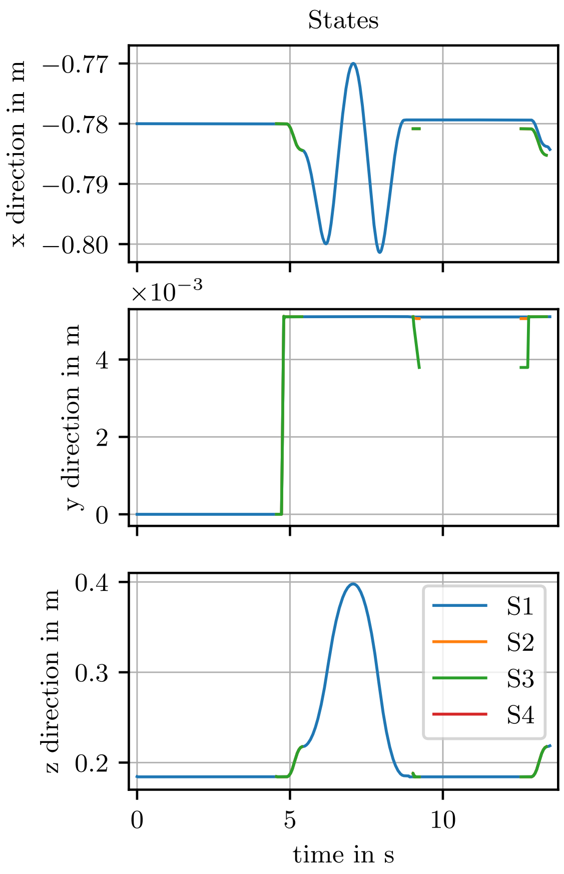


Figure 5. States of the sphere. They are equivalent to the translation of the sphere center in x, y, z direction with respect to its parent. If the sphere is freely moving, world is its parent. States can only be displayed if they are present. If the sphere is rigidly attached to the plate or gripper, there are no states, and nothing is displayed. The sphere in Scenario 4 (S4) has no states. Therefore, nothing is displayed. The states for scenarios 2 and 3 (S2, S3) are available and are displayed if the sphere is freely moving. For the absolute position of the sphere center see Figure 6.

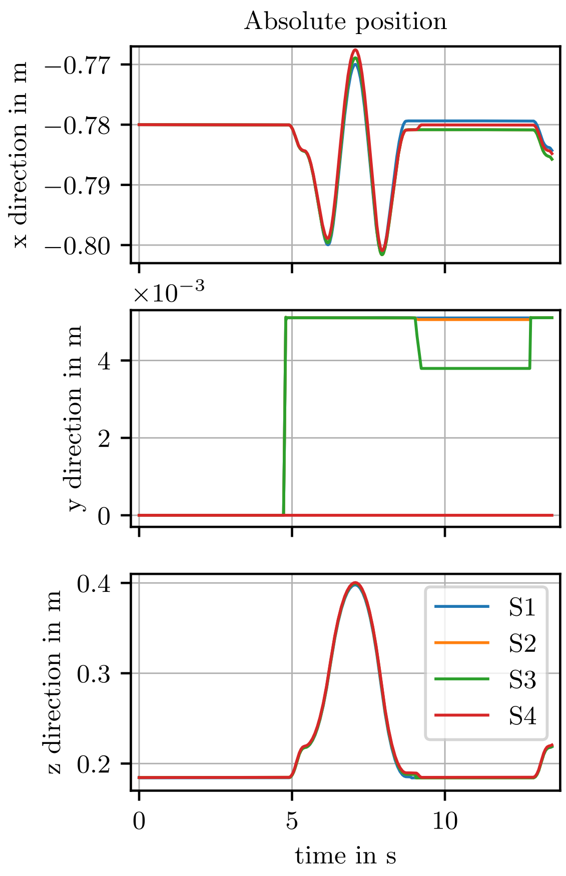


Figure 6. Absolute position of sphere center for the 4 scenarios.

Table 1. Mean $\bar{x}$ and standard deviation s of the simulation time of all four scenarios (S1–S4) each for $n = 12$ runs on a standard notebook[8].

	$\bar{x}$	s
S1	7.816 s	0.123 s
S2	7.255 s	0.075 s
S3	6.863 s	0.388 s
S4	0.397 s	0.016 s

In this section, several combinations of segmented simulation and collision handling are discussed using a KUKA YouBot robot gripping and transporting a sphere, see Figure 4. This robot has a 5 DoF arm and was manufactured in the years 2010–2016. Elmqvist et al. (2021) model the drive trains and controllers of the robot in Modia, and the 3D mechanics with Modia3D.

Four variants of the following transportation scenario are simulated. In all these scenarios, the robot follows the same trajectory. Initially, the cargo, e.g., a sphere, rests on a plate. It is gripped by the robot's gripper and transported upwards until it is placed down again, where it rests on the plate until it is gripped again. The states of the freely moving sphere (see Figure 5), if available, and the absolute position of the sphere center (see Figure 6) are displayed. The simulation times of all four scenarios are compared in Table 1.

Scenario 1 (S1)[9]: The transportation scenario is modeled with collision handling, compare (Neumayr and Otter 2023, Scenario 2(b)). This means, the sphere collides with the plate, as well as with the fingers of the gripper.

Scenario 2 (S2)[10]: The transportation scenario is modeled with segmented simulation and collision handling, see Listing 8. DoFs are added or removed during simulation: At the beginning, the sphere is rigidly attached to the plate. Shortly before the gripper reaches the sphere, the sphere is released (+6 DoF) and collides with the plate. Shortly afterwards it collides with the gripper. After approximately one second, the sphere is rigidly attached to the gripper (-6 DoF). Until the gripper is again close to the plate to release the sphere (+6 DoF), which collides with the plate. Collision handling remains on even if the sphere is rigidly connected to the gripper or plate, as collisions with other bodies can still occur.

Listing 8. Robot program of Scenario 2. Collision handling is enabled by default. It can be turned off or on again in all action commands with `enableContactDetection`. Scenario 3 is defined, by setting this flag to false, as indicated in the comment lines.

```
function robotProgram(actions)
  addReferencePath(actions, ...)

  # 1. attach sphere to plate, -6 DoF
  ActionAttach(actions, "sphereLock",
    "robot.base.plateLock",
    # enableContactDetection = false)

  # 2. some movement of robot
  ptpJointSpace(actions, [
    # open gripper + move to top
    # open gripper + move to plate   ])

  # 3. release sphere off plate, +6 DoF
  # it collides with plate and gripper
  ActionRelease(actions, "sphereLock")

  # 4. gripping via collision handling
  ptpJointSpace(actions, [
    # grip
    # grip + transport a bit          ])

  # 5. attach sphere to gripper, -6 DoF
  ActionAttach(actions, "sphereLock",
    "robot.gripper.gripperLock",
    # enableContactDetection = false)

  # 6. some movement of robot with sphere
```

[8]Intel(R) Core(TM) i7-9850H CPU @ 2.6 GHz, RAM 32 GB

[9]This model can be found in Modia3D, v0.12.2, test/Robot/-ScenarioCollisionOnly.jl.

[10]This model can be found in Modia3D, v0.12.2, test/Segmented/ScenarioSegmentedCollisionOn.jl.

```
    ptpJointSpace(actions, [
      # grip + move to top
      # grip + transport
      # grip + move near to plate
      # open gripper                    ])

    # 7. release sphere off gripper, +6 DoF
    # it collides with plate
    ActionRelease(actions, "sphereLock")

    # 8. some movement of robot
    ptpJointSpace(actions, [
      # open gripper + move to plate  ])

    # repeat step 1. - 8.
    ...
  end
```

Scenario 3 (S3)[11]: The transportation scenario is modeled with segmented simulation and collision handling. Scenario 3 is very similar to Scenario 2, except that collision handling is disabled when the sphere is rigidly connected to the gripper or plate, since in this scenario it is already known that no further collisions will occur. This is deactivated with `enableContactDetection = false` in Listing 8. Basically, this means that the distance calculations between each collision pair is switched off in these phases.

Scenario 4 (S4)[12]: The transportation scenario is modeled with segmented simulation only, compare (Neumayr and Otter 2023, Scenario 2(a)) and Listing 9. Collision handling is switched off for this scenario. This means, the sphere is rigidly attached to the plate, when resting, and rigidly attached to the gripper during transportation. Each time the sphere is rigidly connected to the plate or gripper, the segment is re-initialized. Since the relative velocity and angular velocity between the sphere and the gripper is zero, when the sphere is attached to the gripper or attached to the plate, the physics is correctly modeled under the idealized assumption that gripping time is infinitely small. Basically, this means that gripping effects are neglected.

Listing 9. Robot program of Scenario 4.

```
  function robotProgram(actions)
    addReferencePath(actions, ...)

    # 1. attach sphere to plate
    ActionAttach(actions, "sphereLock",
      "robot.base.plateLock")

    # 2. some movement of robot
    ptpJointSpace(actions, [
      # open gripper + move to top
      # open gripper + move to plate
      # grip                            ])

    # 3. attach sphere to gripper
    ActionAttach(actions, "sphereLock",
      "robot.gripper.gripperLock")

    # 4. some movement of robot
    ptpJointSpace(actions, [
      # grip + transport a bit
      # grip + move to top
      # grip + transport
      # grip + move near to plate
      # open gripper                    ])

    # 5. release sphere off gripper
    # attach it to plate
    ActionReleaseAndAttach(actions,
      "sphereLock", "robot.base.plateLock")

    # repeat step 2. - 5.
    ...
  end
```

[11]This model can be found in Modia3D, v0.12.2, test/Segmented/ScenarioSegmentedCollisionOff.jl.

[12]This model can be found in Modia3D, v0.12.2, test/Segmented/ScenarioSegmentedOnly.jl.

The simulation time of Scenario 4 is about 19 times less than that of Scenario 1. This is because Scenario 4 (segmented simulation only) is basically a non-stiff system where the solver can use large step sizes. In addition, the time for reconfiguration of the multibody system, for gripping and releasing, is negligible. Fine-tuning of collision handling during transportation of the gripped freight is no longer required. Furthermore, any type of cargo can be transported, regardless of its shape. The disadvantage is that the details of the gripping are not modeled, but this can be important.

Scenario 1 (collision handling only) is a stiff system because the gripper holds the sphere by elastic contact and friction forces, which change during transport. Therefore, the solver must use much smaller step sizes. One limitation of collision handling with the MPR algorithm is that it only supports point contacts. If the cargo would be a box, see (Neumayr and Otter 2023, Scenario 3(b)), it would not be possible to calculate a unique point contact that is continuous over time, for example, because one box and one gripper face or one box and one plate face are parallel to each other during contact. All these considerations lead to a compromise in modeling the gripping and releasing of the cargo with collision handling, and otherwise rigidly attaching the sphere to the plate or gripper, resulting in Scenario 2 and Scenario 3.

There is not such a big difference in simulation time for Scenarios 1,2,3, see Table 1. In all three cases, the calculation of the elastic contact response is the limiting factor. This effect is modeled in all these cases. In more realistic scenarios, the approach of Scenario 2 or 3 may pay off, if the number of collision phases is small relative to the remaining actions.

4 Conclusion

The novel approach of variable structure systems with Modia/Modia3D seems to be very promising. Depending of the predefined acausal component and the application one can design (extend) specific actions to trigger new segments and re-initialize the model. In this paper, existing Modia3D actions are extended by enabling or disabling collision handling during simulation, which speeds up the simulation and allows to model form locked fixing of cargos. Furthermore, an example was sketched of how the Modelica language and the FMI standard could be enhanced to allow the number of variables and equations to be changed during simulation.

References

Bezanson, Jeff et al. (2017). "Julia: A fresh approach to numerical computing". In: *SIAM review* 59.1, pp. 65–98. DOI: 10.1137/141000671.

Elmqvist, Hilding (2022). *Slides 7-10 of Modia – A Prototyping Platform for Next Generation Modeling And Simulation Based on Julia. Jubilee Symposium 2019: Future Directions of System Modeling and Simulation.* URL: https://modelica.github.io/Symposium2019/slides/jubilee-symposium-2019-slides-elmqvist.pdf (visited on 2022-12-04).

Elmqvist, Hilding et al. (2021). "Modia - Equation Based Modeling and Domain Specific Algorithms". In: *14th International Modelica Conference*, pp. 73–86. DOI: 10.3384/ecp2118173.

Flores, Paulo et al. (2011). "On the continuous contact force models for soft materials in multibody dynamics". In: *Multibody system dynamics* 25.3, pp. 357–375. DOI: 10.1007/s11044-010-9237-4.

Hertz, Heinrich (1896). *On the contact of solids - On the contact of rigid elastic solids and on hardness. In Miscellaneous papers, MacMillan, 1896, pp. 146–183.* https://archive.org/details/cu31924012500306, accessed on 2023-01-13.

Lenord, Oliver et al. (2021). "eFMI: An open standard for physical models in embedded software". In: *14th International Modelica Conference.* DOI: 10.3384/ecp2118157.

Mattsson, Sven Erik, Martin Otter, and Hilding Elmqvist (2015). "Multi-mode DAE systems with varying index". In: *11th International Modelica Conference*, pp. 89–98. DOI: 10.3384/ecp1511889.

Mehlhase, Alexandra (2014). "A Python framework to create and simulate models with variable structure in common simulation environments". In: *Mathematical and Computer Modelling of Dynamical Systems* 20.6, pp. 566–583. DOI: 10.1080/13873954.2013.861854.

Modelica Association (2022). *Functional Mock-up Interface Specification - Version 3.0.* https://fmi-standard.org/docs/3.0/.

Modelica Association (2023). *Modelica – A Unified Object-Oriented Language for Systems Modeling, Language Specification, Version 3.6.* https://specification.modelica.org/maint/3.6/MLS.pdf, accessed on 2023-06-10.

Neumayr, Andrea and Martin Otter (2017). "Collision Handling with Variable-step Integrators". In: *8th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools.* EOOLT'17. ACM, pp. 9–18. DOI: 10.1145/3158191.3158193.

Neumayr, Andrea and Martin Otter (2019). "Collision Handling with Elastic Response Calculation and Zero-Crossing Functions". In: *9th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools.* EOOLT'19. ACM, pp. 57–65. DOI: 10.1145/3365984.3365986.

Neumayr, Andrea and Martin Otter (2023). "Modelling and Simulation of Physical Systems with Dynamically Changing Degrees of Freedom". In: *Electronics* 12.3. ISSN: 2079-9292. DOI: 10.3390/electronics12030500.

Snethen, Gary (2008). "Xenocollide: Complex collision made simple". In: *Game Programming Gems 7.* Course Technology. Charles River Media, pp. 165–178. ISBN: 978-1-58450-527-3.

Tinnerholm, John, Adrian Pop, and Martin Sjölund (2022). "A Modular, Extensible, and Modelica-Standard-Compliant OpenModelica Compiler Framework in Julia Supporting Structural Variability". In: *Electronics* 11.11, p. 1772. ISSN: 2079-9292. DOI: 10.3390/electronics11111772.

Zimmer, Dirk (2010). "Equation-based modeling of variable-structure systems". PhD thesis. ETH Zurich. DOI: 10.3929/ethz-a-006053740.